7-1-2013

Understanding and modeling the synchronization cost in the GPU architecture

James Letendre

Understanding and Modeling the Synchronization Cost in the GPU Architecture

by

James T. Letendre

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science
in Computer Engineering

Supervised by

Dr. Sonia Lopez Alarcon
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
July 2013

Approved by:

Dr. Sonia Lopez Alarcon
Department of Computer Engineering

Dr. Marcin Lukowiak
Department of Computer Engineering

Dr. Roy Melton
Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Understanding and Modeling the Synchronization Cost in the GPU Architecture

I, James T. Letendre, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

James T. Letendre

Date

Dedication

I dedicate this thesis to my family for supporting me over the years.

Acknowledgments

I would like to thank my advisors for their support and guidance with this work. I would also my friends and family for their support.

Abstract

Understanding and Modeling the Synchronization Cost in the GPU Architecture

James T. Letendre

Graphic Processing Units (GPUs) have been growing more and more popular being used for general purpose computations. GPUs are massively parallel processors which make them a much more ideal fit for many algorithms than the CPU is. The drawback to using a GPU to do a computation is that they are much less efficient at running algorithms with more complex control flow. This has led to them being used as part of a heterogeneous system, usually consisting of a CPU and a GPU although other types of processors could be added.

Models of GPUs are important in order to determine how well your code will perform on various different GPUs, especially those which the programmer does not have access to. GPU prices range from $100s to $2000s and more, so when designing a system with a particular performance value in mind, it is beneficial to be able to determine which GPU best meets your goal without wasting money on unneeded performance.

Current GPU models were either developed for older generations of GPU architectures, they ignore certain costs that are present in the GPU, or when they account for those costs, they do so inaccurately. The big component that is ingored in most of the models investigated is the synchronization cost. This cost arrises when the various threads within the GPU need to share data amongst themselves. In order to ensure that the data shared is accurate, the threads must synchronize so that they have all written to memory before any thread tries to read. It is also the cause of major inaccuracies with the most up to date GPU model found. This thesis aims to understand the factors of the synchronization cost through the use of microbenchmarks. With this understanding the accuracy of the model can be improved.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Much work has been done into modeling the runtime of CUDA kernels on the various NVIDIA architectures; however, most of these models do not account for the synchronization cost, or do so poorly. Depending on the kernel, the effect of ignoring the synchronization cost may be minimal; however, if the model is used when a kernel will run many times, the small error will compound and become an issue.

There have been many models for the GPU developed since the introduction of CUDA [4, 6, 8, 9]. Some of these models are outdated due to changes in the GPU architecture made by NVIDIA over the years, such as the improved coalescing model with GT-200 and the addition of L1/L2 cache to global memory in the Fermi architecture. One thing in common among almost all of these models is the fact that the synchronization cost is entirely ignored when computing the runtime for the kernel. The model in [8] does attempt to account for the synchronization cost, but they do so with an equation that tends to over estimate the cost by a large margin in some circumstances. When trying to verify one set of results from the paper, using reduction kernels from the NVIDIA CUDA SDK examples, a large discrepancy was found mostly attributable to the synchronization cost. While some kernels might not need to have any synchronization barriers in them, if any data need to be shared among threads during the computation, then a synchronization will be necessary.

If the model is to be used in something such as simulating scheduling of a program on a heterogeneous system (CPU+GPU or CPU+GPU+FPGA) then the error due to the missing synchronization cost will compound over each kernel run. While this cost may initially be small, given enough kernel

executions, the cost will grow to a substantial amount. This can result in a task being scheduled on a different processor type because the simulator would think, incorrectly, that the GPU is still busy. This is an issue because some computations run significantly faster on the GPU than they run on the CPU. GPUs are much better at running highly parallel computations than a CPU is, and scheduling one of these on the CPU would throw the rest of the simulation off by adding extra delays.

To develop a model for the synchronization cost, we ran micro-benchmarks for various aspects that could affect the synchronization cost such as global and shared memory accesses. Micro-benchmarks are small programs that are designed to test a single aspect of a system rather than the system in general. The data obtained from these micro-benchmarks show some interesting features which give insight into what launch parameters would result in more optimal performance for a kernel where synchronization costs are a factor. The features also help to show some of the internal characteristics of the GPU. Using these new understandings of the GPU and the factors that affect the synchronization cost, the accuracy of the model can be improved over what was originally presented. These equations, while intended to replace some of those presented in [8] could also be used in other models to help improve accuracy.

Chapter 2

Background

2.1 CUDA

CUDA is a widely used library for running applications on NVIDIA GPUs. GPUs are single instruction multiple data (SIMD) systems, running many threads in parallel with no communication between them, such as, a nearest neighbor based simulation, where the next value of a cell depends only on the values of the cells around it. Each time the kernel is run, there can be a large amount of work for it to perform in order to compute the next state, but the work in each cell is independent. However, for some applications this communication is unavoidable, (e.g. , a reduction operation on an array) where the data for each iteration must be shared between the threads before the next iteration can be performed.

2.1.1 Programming Model

NVIDIA GPUs are composed of a number of stream multiprocessors (SM), where each SM is capable of running some number of blocks which is determined by hardware limits. Figure 2.1(a) shows what an SM looks like in the Fermi architecture. In the newer Kepler architecture, the SMs are now called SMXs to differentiate between the two. Figure 2.1(b) shows the SMX used in the newer Kepler architecture of GPUs. In the Figures, each "core" is a stream processor (SP) which is capable of running an instruction for one thread at a time. As can be seen, the Kepler SMX has far more SP units than the Fermi SM does, which means that Kepler cards are capable of performing many more operations in each cycle than Fermi. The Fermi

(a) One stream multiprocessor in the Fermi architecture.

(b) One stream multiprocessor in the Kepler architecture.

Figure 2.1: Comparison of Fermi and Kepler stream multiprocessors.

architecture has only 16 load/store units while Kepler has 32, and, since in CUDA the smallest executable unit is the warp consisting of 32 threads, this means that Kepler is able to schedule an entire warp for memory accesses at the same time, while Fermi must schedule the warp in two halves.

CUDA uses a programming model which consists of threads, blocks, and grids, as show in Figure 2.2. Each kernel launch is a one grid, and each grid is composed of some number of blocks determined by the programmer. The blocks, in turn, are made of some number of threads. The threads when executed are run as sets of 32 called a warp which allows the scheduler to increase the throughput by scheduling more than one thread at a time. Each block is guaranteed to be scheduled within the same SM on the GPU. This

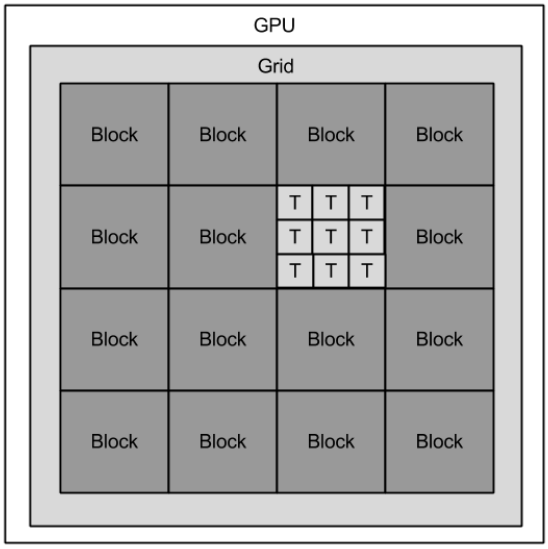


Figure 2.2: CUDA programming model.

is important because in cases where data must be shared this ensures that all the threads within the same block have the same local L1 cache, so the memory read associated with sharing the data has the potential to be faster than if it were forced to go to the globally shared L2 cache always. Each SM can schedule fixed maximum number of threads, for Fermi this number is 1536, so in order to get the maximum performance out of the GPU, it is important to make sure that the chosen block size would allow each SM to be used to as close the 100% capacity as possible.

All threads in the CUDA kernel run the exact same program, but they need to be able to determine which thread number they are in order to operate on the correct elements from memory. To accomplish this, CUDA provides a few built-in global variables: threadIdx, blockIdx, blockDim, and gridDim. The variables with the "Idx" suffix give either the thread's index within its respective block or the block's index within the grid, respectively, whereas the variables with the "Dim" suffix provide the number of threads in each block and number of blocks in the grid, respectively. Generally, only the first three are used which allow the global ID to be established; gridDim would be used only if the total number of threads launched was needed. The ID can then be used to determine which location in memory this thread

```
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <cuda_runtime.h>

/*
 * An example CUDA kernel
 */
__global__ void example_kernel( float* data )
{
        // The index of the current thread
        int idx = threadIdx.x + (blockDim.x * blockIdx.x);

        // write our ID to the array
        data[idx] = idx;

        return;
}
```

Listing 2.1: Sample CUDA kernel

should be responsible for processing. A sample CUDA kernel is shown in Listing 2.1. This kernel is very simple only writing its ID into the data array passed into the kernel; however it shows how the ID can be determined and used within a kernel.

A more complicated example is shown in Listing 2.2. This kernel uses shared memory to avoid multiple reads from global memory; it also shares data between the threads each iteration. In cases such as this, it is necessary to ensure that data dependencies are correctly being observed. This is accomplished by use of the `__syncthreads()` function, which causes all threads in the same block to wait until all the other threads in the block also reach a `__syncthreads()` function call.

2.1.2 Code Compliation

Code written in CUDA must be compiled into a form that the GPU can understand and execute just like code written for any other system. In the GPU this machine code is known as CUBIN; however there are two different final products the compiler can produce, PTX and CUBIN. PTX code is a pseudo-assembly language as it does not relate 1-to-1 to hardware instructions. It is used by CUDA as a first pass stage during the compilation, and

```
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <cuda_runtime.h>

/*
 * An example CUDA kernel utilizing __syncthreads()
 */
__global__ void sync_kernel( float* data )
{
    float sdata[];

    // load shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

        // load our value into shared memory
    sdata[tid] = (i < n) ? data[i] : 0;

        // wait for everyone to finish
    __syncthreads();

    // do reduction in shared mem
    for (unsigned int s=1; s < blockDim.x; s *= 2)
    {
        if ((tid % (2*s)) == 0)
        {
            sdata[tid] += sdata[tid + s];
        }

                // again, wait for everyone to finish
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) data[blockIdx.x] = sdata[0];

        return;
}
```

Listing 2.2: Sample CUDA kernel with synchronization

it is more generic in that it is able to be compiled at runtime into whichever version of CUBIN is needed for the particular GPU. PTX is useful as a way to "future-proof" a program to ensure that it will also be able to run on GPUs that are released in the future. CUBIN is the actual assembly language that the GPUs can understand, but it is less flexible than PTX as it only applies to GPUs which are of the specific compute version. NVIDIA uses the compute version of the GPU as a means to differentiate between different capabilities that the GPUs may have. The original CUDA cards were compute version 1.0. Each subsequent architecture upgrade that added new features resulted in a minor version upgrade, such as version 1.3 which added support for double-precision floating-point numbers. Major architecture upgrades such as the Fermi to Kepler architecture result in a major version number upgrade, version 2.x to version 3.x. Each of these different versions have different capabilities and therefore each has a slightly different version of CUBIN that it is able to run. The distinction between PTX and CUBIN is important to understand when analyzing the structure of a CUDA program at the assembly level.

2.1.3 Synchronization Cost

In CUDA a `__syncthreads()` function call causes threads within a block that reach the function to wait until all other threads in the same block also reach a `__syncthreads()` call. This is used to ensure that any data that need to be written to memory and be shared among the threads in a block will have actually been written before another thread attempts to read it. Algorithms that fall under the "embarrassingly parallel" category, where all operations can be performed perfectly in parallel without any communication between them, have no need for the `__syncthreads()` function as no data need to be shared. However, most algorithms require some communication between threads and therefore need to synchronize their execution at those points.

The time spent waiting to synchronize is called the synchronization cost which is shown in Figure 2.3. As is evident in the figure, each warp experiences slightly different synchronization costs. The first thread to reach the

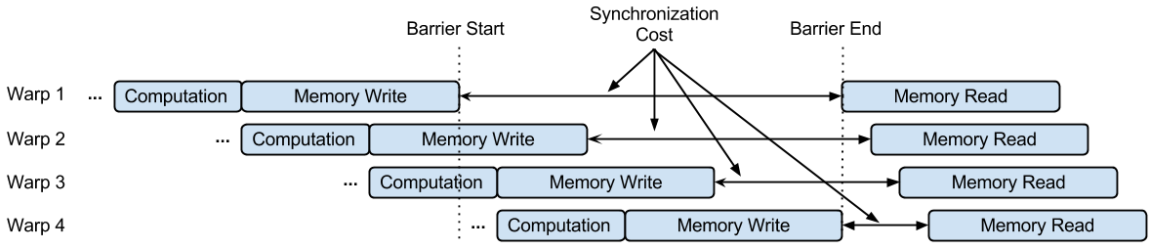


Figure 2.3: Visualization of the synchronization cost.

barrier experiences the largest cost, while the last thread to reach the barrier experiences the minimum cost. The *maximum cost* is mainly driven by the memory accesses which occur before the synchronization, while the *minimum cost* is driven mainly by the delay imposed by having to schedule all the warps in the block at the same time.

In Figure 2.1(a), note the two scheduling units near the top of the SM; each of them has two instruction dispatch units. The GPUs run threads in groups of 32, called a warp, to increase performance. For Fermi, each schedule can dispatch new warps every other cycle, so each SM can effectively schedule two warps per cycle. Unless the kernel being run uses a very large amount of shared memory or registers, there will most likely be more than four warps running on each SM at any given time. The scheduling limits will result in small differences among each thread and where it is in the execution of the kernel since only two of the warps are able to advance each cycle. This is the main driver behind the minimum synchronization cost shown in Figure 2.3.

Referring to Figure 2.1(a) again, note that each SM has only 16 load/store units. Given the warp size of 32 threads, this means that only half of each warp can access memory at any given time. Not only does each warp have to wait internally until all threads have read their respective memory, but warps have to wait for the other warps in the same block when they reach a `__syncthreads()` call. Given that the main reason that `__syncthreads()` would be called is to ensure that memory is read only after it has been written, this cost will always be present in the synchronization cost.

The main cost that contributes to the maximum synchronization cost is

the deviation that comes from reading from or writing to memory. When the kernel gets to the point where it needs to read a value from memory, odds are that all, or most, of the threads in the kernel are also going to need to read something from memory. If they all require the same value, such as a parameter to the function, then only one memory read is performed, and the result is given to all threads. However, the bulk of the work on a GPU happens when each thread reads different values from memory and then performs the same operations on those values before writing them back into memory. This is why GPUs are so good at performing parallel computations, because it can do the operation in parallel; however, because all of the threads need to access memory at essentially the same time, and all of them need to read from somewhere different, this causes the threads to diverge around the memory access. The threads that get their data first are able to start their computations, while other threads are still waiting for data. The maximum synchronization cost is dominated by that difference in memory access times.

There are many factors that affect how large of a difference there will be between the first and last thread to read memory. A big factor is the occupancy of the kernel, how close to the maximum number of threads are running at the same time. There are memory factors also, such as which type of memory is being read from and how many bytes each thread needs to read.

2.2 GPU Model

In this section, we discuss the model presented in [8]. This paper improves on the work presented by one of the authors in [4], which presents their memory warp parallelism-compute warp parallelism (MWP-CWP) model. The original model was developed for the GT-200 architecture, and with their new work they update the model to the Fermi architecture. They also make the model easier to apply by abstracting away many of the terms and parameters that were required to be supplied in the previous work.

The main concept in the model is the values of CWP and MWP. MWP measures how many outstanding requests to memory there can be at any

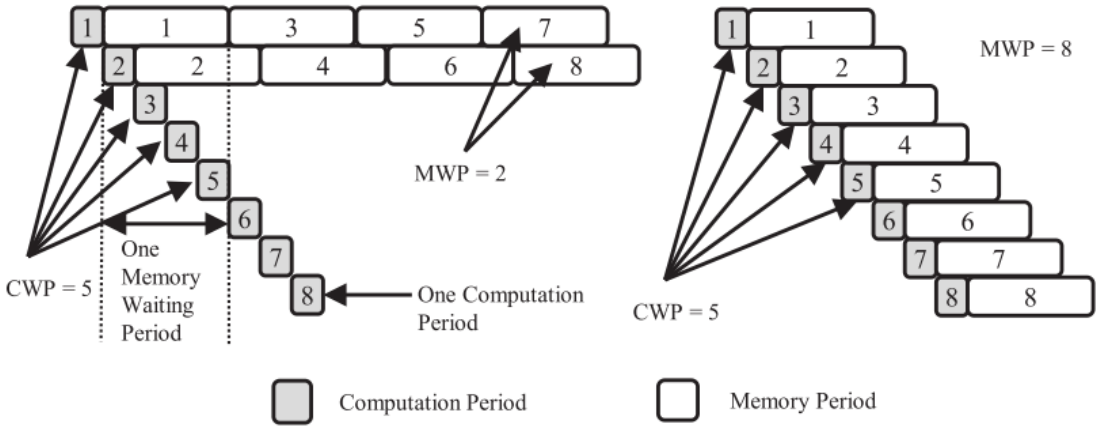


Figure 2.4: The effect of the values of MWP and CWP on the kernel runtime.

given time. The CWP is the measure of how many compute instructions are required to hide the latency associated with a memory access plus one, (i.e. , the instruction that generates the memory access plus how many would be needed to hide that memory access). The model, like most others, splits the memory and computation times into separate factors. The relation between MWP and CWP determines whether memory or computation cost is the main component of the total execution time.

1. $MWP < CWP$: The cost of computation is hidden by the cost of memory operations. The total execution time is determined mainly by the memory cost. This is shown in the left half of Figure 2.4.

2. $MWP >= CWP$: The cost of memory operations is hidden by the cost of computations. The total execution time is determined mainly by the computation cost. This is shown in the right half of Figure 2.4.

3. *Not enough warps*: There are not enough warps to completely hide either cost. The total execution time is determined by some amount of both costs.

The model has several stages that it undergoes to extract the various parameters needed by the model equations. The first is using the NVIDIA Compute Visual Profiler which is able to extract various parameters during the execution of a kernel, (e.g. , the cache miss rates, which are difficult

to determine from static analysis of the code). The second stage is static analysis of the code, this occurs in two parts: PTX analysis and CUBIN analysis.

2.2.1 PTX analysis

The GPU model makes use of GPGPU Ocelot [3] to perform analysis on the compiled PTX code from the kernel. GPGPU Ocelot is a PTX emulator, emulating the GPU for whichever CUDA kernel is passed to it. Because the emulation is being done in software, some extra analysis can be done that is not possible on the actual GPU hardware. The first thing that Ocelot does when given a kernel to run, is to break the PTX code up into the basic blocks that make up the program. A basic block is defined as a block of continuous code that has only one entry point and only one exit point. This means that once a block starts executing, it must be executed to the end of the block, and that there is no way for the block to begin execution anywhere other than the initial instruction.

The basic blocks are found by first marking the first and last instruction as the start and end of a block respectively. Then, branch instructions are marked as the end of a block, and their targets as the beginning of a new block. Finally, every end of block must be followed by the start of a new block, and each new block must be preceded by the end of the previous one; this fills in any gaps that might be left in the structure. For Ocelot, this structure is then used to emulate the PTX code.

While the emulation starts, Ocelot allows for certain tracers to be attached to the kernels and output data about their execution. Among these is the Basic Block Counter, which keeps track of how many times each basic block was executed in the kernel. These data are combined with the data obtained from the CUBIN compiled kernel to obtain more accurate representation of the kernel's parallelism.

2.2.2 CUBIN Analysis

CUDA kernels are written in C, but can be compiled to PTX code, CUBIN code, or both, which are then included in the executable. CUBIN is the device dependent assembly code used by NVIDIA GPUs. Each compute version has its own slightly different version of CUBIN. Some of this is due to the changes in the architecture between generations of GPUs, (i.e. the addition of cache to global memory between GT-200 and Fermi), which allow for more advanced functionality to be exposed.

PTX is the device independent pseudo-assembly code used by NVIDIA. Where CUBIN code is specific to particular GPU architectures and compute versions, PTX is independent of all that. However it is not capable of being run on the GPUs natively; it first must be compiled into CUBIN code which can then be run. This additional compilation step allows NVIDIA to perform additional optimization steps that are particular to each architecture; because of this PTX and CUBIN code do not match up 1-for-1.

In [8], the authors developed some custom code to parse through the CUBIN code, which was disassembled with $cuobjdump$, and extract the basic blocks. This code was reproduced for this work. This stage is also where the instruction level parallelism (ILP) and memory level parallelism (MLP) parameters for the model, as well as various instruction counts, are extracted.

ILP is determined by taking each basic block extracted from the CUBIN and building a map to where each register is written and from where each register is read. The basic block is then broken into groups of instructions such that within each group no register is read by an instruction after being written by an earlier instruction. Essentially, each group is a set of instructions which could be scheduled simultaneously with no data dependency issues.

The value of ILP is given in (2.1). As can be seen in the equation, whenever there is a basic block such that every instruction uses the result of the instruction before it, then N_{groups} will be equal to $N_{instructions}$ since each instruction would be in its own group. This would result in a value for ILP of 1 which means that there is no level of parallelism present in the basic block and that it is instead completely sequential. Conversely, if

every instruction in a block is independent of the others, they would all be in a single group, which would make ILP equal to $N_{instructions}$ meaning that each of the instructions could be scheduled at the same time with no issues, (i.e. the block is completely parallel).

$$ILP = \frac{N_{instructions}}{N_{groups}} \tag{2.1}$$

The value for MLP is calculated in a similar fashion. This time the value of interest is the parallelism of the memory instructions, so a map of what registers are loaded from memory and when they are later accessed is built. MLP for a single memory operation is defined as the number of memory instructions present between the memory instruction which loads a value and the instruction which uses the value, (i.e. when the value is actually needed to have been read). The value for MLP is given in (2.2).

$$MLP = \frac{\sum_{i}^{mem\_instrutions} MLP_{instruction}(i)}{N_{mem\_instructions}} \tag{2.2}$$

Once again, if each memory access is immediately followed by an access to its variable, then there is no parallelism present in the memory accesses, and consequently the MLP is 0. When all the memory access are grouped together without accesses to their variables in between, then all the memory accesses in the block can be performed in parallel. In this case the MLP is more complex than the ILP value for the parallel case; however it will still be at its maximum value.

Once the PTX and CUBIN data are obtained, the basic blocks can be roughly matched up between the two versions of the code. The structure will not match exactly due to the extra optimizations that occur between PTX and CUBIN during the compilation; however, the basic algorithm structure will not change. Loops will still be present in both versions, and branches that diverge into two possible paths will also be present. The important match-up that needs to happen is the block counts from GPGPU Ocelot onto the CUBIN blocks, mainly for any loops, but also useful for branches. These are used to weight the ILP and MLP values for each block to come

up with the kernel ILP and MLP. The equation for the kernel ILP is given in (2.3); the equation for the kernel MLP follows the same form.

$$ILP_{kernel} = \frac{\sum_{i=0}^{N_{blocks}} Accesses(Block_i) * ILP(Block_i)}{N_{blocks}} \tag{2.3}$$

2.2.3 Model Equations

Table 2.1 gives a list of parameters used in the model with their respective descriptions.

Table 2.1: Descriptions of model parameters.

Name	Description
N_{insts}	Number of instructions executed per warp.
N_{mem_insts}	Number of memory instructions per warp.
N_{SFU_insts}	Number of Special Function Unit (SFU) instructions per warp.
N_{sync_insts}	Number of synchronization instructions per warp.
N_{total_warps}	Total number of warps executed in the kernel.
N_{active_SMs}	Number of active SMs while running the kernel.
$N_{warps\_per\_SM}$	Number of warps running on each SM at once.
$AMAT$	Average memory access time.
avg_trans_warp	Average number of memory transactions generated per each memory instruction.
avg_inst_lat	The average latency of instructions in the kernel.
$ITILP$	Inter-thread Instruction Level Parallelism.
$ITILP_{max}$	The maximum ITILP possible given GPU parameters.
$ITMLP$	Inter-thread Memory Level Parallelism.
avg_DRAM_lat	The average latency for memory accesses that must access DRAM.

Continued on next page.

Table 2.1 – continued from previous page

Name	Description
$miss\ ratio$	The miss ratio of caches in global memory.
$hit\ lat$	The latency associated with a hit to cache.
MWP_{cp}	The MWP value bounded by the CWP.
$DRAM\_lat$	The latency of an access to DRAM for this GPU.
Δ	The delay between consecutive results from memory accesses.
$warp\_size$	The number of threads in a warp.
$SIMD\_width$	The number of SPs per SM.
SFU_width	The number of SFUs per SM.
$freq$	GPU shader frequency.
$mem\_peak\_bandwidth$	The peak memory bandwidth available.
Γ	Machine dependent parameter for synchronization cost.
T_{exec}	Total execution time of the kernel, in cycles.
T_{comp}	Total computation time of the kernel, in cycles.
T_{mem}	Total memory access time of the kernel, in cycles.
$T_{overlap}$	Overlap time between computation and memory costs.
$W_{parallel}$	Time to issue all instructions in each SM.
W_{serial}	Serial costs incurred by instructions. Synchronization cost applies here.
O_{branch}	Overhead due to branch divergence.
O_{bank}	Overhead due to bank conflicts in shared memory.
O_{SFU}	Overhead due to contention over the SFU units.
O_{sync}	Overhead due to synchronization instructions.

The main equation for the model is simple. The total execution time is the time spent issuing and performing instructions plus the time spent doing memory accesses, less any overlap between the two values. Overlap can play a rather large role in the execution time of a kernel; if there is

significantly more computation than memory, the memory time could be mostly hidden.

$$T_{exec} = T_{comp} + T_{mem} - T_{overlap}$$

Computation cost

The equations involved in computing T_{comp} and T_{mem} are obviously more complex than this. The computation time is broken into $W_{parallel}$ and W_{serial} which are the costs due to issuing all the instructions in the kernel, and the cost of any waiting that is introduced by those instructions, respectively.

$W_{parallel}$ is the cost incurred from having to issue all of the instructions in the kernel. The first term is the total number of instructions in each of the active SMs, and the second is the effective throughput of instructions. $ITILP$ is the inter-thread instruction level parallelism; this term measures not just how parallel the kernel is internally, but how well it can be parallelized amongst the other warps running the same code.

$$W_{parallel} = \frac{N_{insts} \times N_{total\_warps}}{N_{active\_SMs}} \times \frac{avg\_inst\_lat}{ITILP}$$

$ITILP$ is computed as shown below, where ILP for the kernel is computed as shown previously. The ideal case is that the kernel $ITILP$ will be limited by the average instruction latency, (i.e. there is more than enough parallelism present in the kernel to fill the gaps while instructions for each warp execute). When this is the case, the effective throughput for the kernel instructions becomes 1, meaning that there will always be an instruction to schedule whenever it is needed, so the parallel cost would be equal to the number of instructions that need to be scheduled. At worst case, the $ITILP$ will instead be limited by the ILP of the kernel. $N_{warps\_per\_SM} \times ILP$ gives the level of parallelism the kernel is able to provide on each SM. In this case the effective throughput value becomes greater than 1, which causes the parallel cost to be greater than just the number of instructions present in the kernel, but also accounts for the gaps that will appear when there are no instructions to schedule.

$$ITILP = min(N_{warps\_per\_SM} \times ILP, ITILP\_max)$$
$$ITILP\_max = \frac{avg\_inst\_lat}{warp\_size/SIMD\_width}$$

Next, we have W_{serial} which measures the various costs for times where the kernel is required to stop and wait on something. These things include branch divergence, memory bank conflicts, SFU instruction contention, and synchronization cost.

$$W_{serial} = O_{branch} + O_{bank} + O_{SFU} + O_{sync}$$

O_{branch} and O_{bank} are complex costs that are not accounted for in the model. Instead they must be provided to the model as a cycle cost by the programmer. O_{SFU} and O_{sync} on the other hand are covered in the model.

O_{SFU} represents the contention over the limited number of Special Function Units in each SM. If many SFU instructions are issued too closely together, the kernel will need to wait for the preceding instructions to finish before the next can be issued because the SFU takes longer to complete the more complex operations it performs than normal instructions take to complete. The SFU instructions cost is mostly dependent on the number of SFU units available and how many SFU instructions are present in the kernel relative to the other instructions.

$$O_{SFU} = \frac{N_{SFU\_insts} \times N_{total\_warps}}{N_{active\_SMs}} \times \frac{warp\_size}{SFU\_width} \times F_{SFU}$$
$$F_{SFU} = min\left\{max\left\{\frac{N_{SFU\_insts}}{N_{insts}} - \frac{SFU\_width}{SIMD\_width}, 0\right\}, 1\right\}$$

F_{SFU} depends on the ratio of SFU instructions present in the kernel, and the ratio of how many SFU units there are in each SM to the number of threads in each warp. If the ratio of SFU instructions in the kernel is less than the ratio of SFU units to threads then there will be no contention on the hardware units. On the other hand if the ratio of SFU units to threads is less

than the ratio of instructions in the kernel, then there is more demand for the units than can be supplied by the SM, so the factor will be greater than 0, meaning there will be delays. O_{FSU} then takes this factor and scales by the number of SFU instructions in the kernel, as well as the ratio of threads in a warp to the number of hardware units. This allows the cost to scale up if the SFUs are required to be used multiple times to complete one instruction for a all threads in a warp.

O_{sync} represents the cost incurred by the `__syncthreads()` function calls. This cost is present in kernels where data need to be shared between threads because proper execution requires all of the blocks threads to be at the same place in the kernel to prevent data from being read before it has been by a thread upon which it depends. In the model the authors assume that this cost is a constant that mainly depends on the memory access time. In the equations below Γ is a "machine dependent parameter" that the authors determined to have a value of 64.

$$O_{sync} = \frac{N_{sync\_insts} \times N_{total\_warps}}{N_{active\_SMs}} \times F_{sync}$$
$$F_{sync} = \Gamma \times avg\_DRAM\_lat \times \frac{N_{mem\_insts}}{N_{insts}}$$

F_{sync} is the cost of each synchronization. In the model, this cost depends only on the DRAM latency and the ratio of memory instructions to total instructions.

Memory Cost

The memory cost is the other main cost in the system aside from the computation cost, and it is given as follows.

$$T_{mem} = \frac{N_{mem\_insts} \times N_{total\_warps}}{N_{active\_SMs} \times ITMLP} \times AMAT$$

The first term is the effective number of requests made per SM. It is the total number of memory instructions present in each SM divided by the inter-thread Memory Level Parallism ($ITMLP$). The $ITMLP$ is similar to

the $ITILP$ in that it represents the memory level parallelism for the entire kernel rather than just within the kernel. $AMAT$ is the typical definition: it represents the average memory access time for all memory operations including hits/misses to cache.

$$ITMLP = min\left(MLP \times MWP_{cp}, MWP_{peak\_bw}\right)$$
$$MWP_{cp} = min\left(max\left(1, CWP - 1\right), MWP\right)$$

$$AMAT = avg\_DRAM\_lat \times miss\_ratio + hit\_lat$$
$$avg\_DRAM\_lat = DRAM\_lat + (avg\_trans\_warp - 1) \times \Delta$$

Here hit_lat and $miss\_ratio$ account for any levels of cache on the global memory.

Overlap Time

The final cost in the system is the overlap time. This is the amount of time the GPU is able to perform calculations from other warps while it is waiting on memory accesses from others. Ideally there would be complete overlap of all memory operations in the kernel; however this is not always the possible with some algorithms. The amount of overlap depends on the CWP and MWP of the particular kernel. If the CWP is higher than the MWP then the kernel will be able to cover the cost of memory accesses with computation from other warps. If the CWP is less than or equal to the MWP there will not be enough computations to completely cover the memory accesses. How much of the memory cost is exposed is determined by how many warps there are on each SM.

$$T_{overlap} = min\left(T_{comp} \times F_{overlap}, T_{mem}\right)$$
$$F_{overlap} = \frac{N_{warps\_per\_SM} - \zeta}{N_{warps\_per\_SM}}$$
$$\zeta = \begin{cases} 1, & (CWP \leq MWP) \\ 0, & (CWP > MWP) \end{cases}$$

CWP and MWP

The computation for CWP and MWP is taken from the previous work on the model [4]. CWP is the measure of how many computations are required to cover the cost of one memory operation plus one.

$$CWP = min\left(CWP\_full, N_{warps\_per\_SM}\right)$$
$$CWP\_full = \frac{mem\_cycles + comp\_cycles}{comp\_cycles}$$
$$comp\_cycles = \frac{N_{insts} \times avg\_inst\_lat}{ITILP}$$
$$mem\_cycles = \frac{N_{mem\_insts} \times AMAT}{MLP}$$

mem_cycles is how many cycles each warp is waiting for memory accesses. $comp\_cycles$ is how many cycles each warp spends doing computation. CWP_full is the unbound CWP, the max value CWP can have is equal to the number of warps running on the SM since that is the number of warps the SM has to work with.

MWP is the measure of how many outstanding memory requests there can be at maximum.

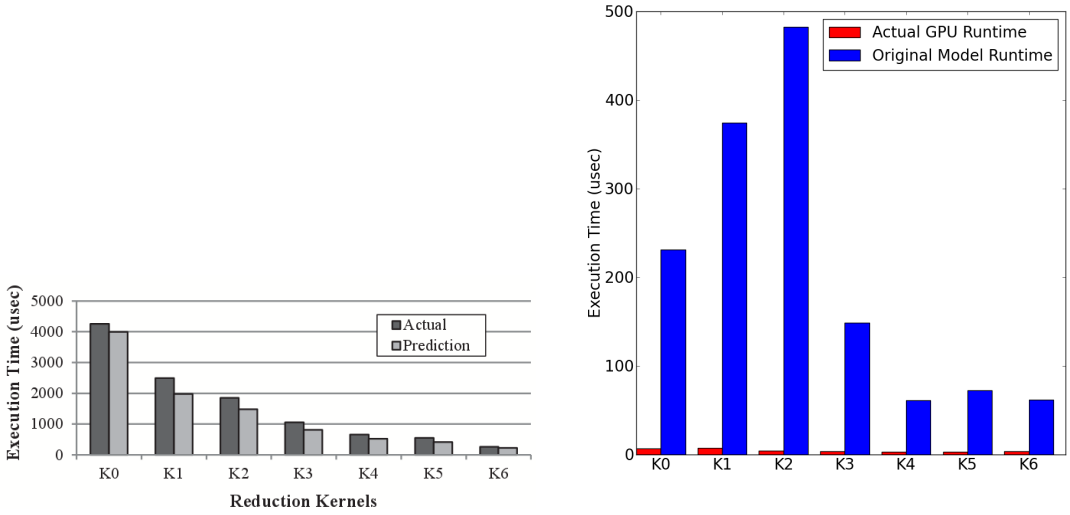
$$MWP = min\left(\frac{avg\_DRAM\_lat}{\Delta}, MWP_{bw}, N_{warps\_per\_SM}\right)$$
$$MWP_{bw} = \frac{mem\_peak\_bandwidth}{BW_{warp} \times N_{active\_SMs}}$$
$$BW_{warp} = \frac{freq \times transaction\_size}{avg\_DRAM\_lat}$$

$transaction\_size$ is how many bytes of memory can be read in each memory transaction. $freq$ is the clock speed of the GPU. BW_{warp} is the amount of bandwidth that each warp requires to read all of its data from memory. $mem\_peak\_bandwidth$ is the maximum bandwidth between the SM and DRAM.

2.2.4 Original Model Issues

The model, while rather complete in all of the costs it tries to account for, is not very accurate with some of the more complex costs, like synchronization. In [8], one of the examples used to show how the model works are the reduction kernels present in the NVIDIA GPU SDK. The SDK contains seven kernels which all perform a partial reduction on an array of numbers. Each kernel starting at reduce0 and going to reduce6, moves from the naive, first-pass approach to a much more heavily optimized version. This is done as an example of steps that can be taken in a kernel to increase performance. In the model, it serves as a nice example for how well the model is able to detect and account for some of the various optimizations used with CUDA kernels to improve performance.

The model provides a nice graph comparing the actual runtime of the kernel with their modeled runtime. When we attempted to reproduce the results to ensure that we got similar results, the numbers we got were substantially different from the actual runtime, shown in Figures 2.5(a) and 2.5(b). After verifying that the model code we implemented was indeed correct, we were able to determine that the synchronization cost was a major factor in the error the model was producing.



(a) Reduce model vs runtime from paper.[8]

(b) Reduce model vs runtime obtained from implemented model

Figure 2.5: Comparison of the expected and achieved values for the reduction kernels.

As can be seen, while their numbers show the model as being relatively accurate, the numbers we obtained when we applied the models to the same code give largely inaccurate values. They did not provide details for what size of kernel they ran, so the actual run times that we tested against were different than in the paper; however the error would probably only grow with the actual runtime. We later found out that the equation for the synchronization cost had evolved over time. This may mean that the final equation presented in the paper differs from the one used for the reduction kernels.

Chapter 3

Experimental Setup and Methodology

3.1 Benchmarks

Since the synchronization cost is the result of needing to share memory between threads in a block, the micro-benchmarks were developed to ensure that two main sets of memory were tested. The first set of memory is normal global memory; this memory is the data stored in DRAM with a shared L2 cache among all the SMs and a local L1 cache within each SM. The second set of memory that is commonly used is shared memory. This set of memory shares its space with the L1 cache for global memory; by default the L1 cache occupies 16KB while the shared memory occupies 48KB. The shared memory is explicitly managed by the programmer unlike the L1 cache which is handled by the hardware. For this thesis, four micro-benchmarks were written, two for each type of memory. One of them tested the effects of performing a read from memory, and the other tested the effects of writing to the memory.

There are two parameters that the programmer can control in the launch of a kernel. The first is the number of blocks that are launched on a particular kernel. The second is the number of threads each block contains. When running a kernel to perform a computation, generally the total number of threads is chosen so that each thread has one data element to process. The number of threads per block is chosen to ensure that each block spends as little time idle as possible. For practical purposes the number of threads per block is usually chosen such that it is an even multiple of the warp size for the GPU, which is 32. When testing synchronization costs within a block, if a block has only one warp, 32 threads per block, then there is no other

warp with which to synchronize with and therefore no synchronization cost. The maximum number of threads per block that can be launched is 1024. This limit is different from the limit of maximum number of threads each SM can schedule. It is a limit imposed by the CUDA API. Given these two constraints, kernels were launched to test each viable number of threads per block: 64, 96, 128, . . . , 1024.

The number of blocks range was chosen so that a wide range of possible kernel launches would be covered. Starting with a low of 10 blocks launched, the number of blocks increases up to 2000 blocks launched. At 2000 blocks the data from all benchmarks has reached a steady final trend, be it constant or a slightly increasing value. The number of blocks is increased slowly at first while the synchronization cost data was the most volatile, and as the change in synchronization cost began to become smoother, the number of blocks launched was increased faster. This ensures that features that occur at low numbers of blocks launched can be accurately captured while reducing the overall runtime of the benchmarks by using lower resolution where the synchronization cost data is smoother.

Listings 3.1 and 3.2 show examples of two of the micro-benchmarks used in this thesis. The first shows the benchmark used to test reading from shared memory. Note that while nothing is written into shared memory before reading it, the compiled code still executes the shared read instruction to read the data from shared memory, and since the value is not used for any real purpose in the kernel, reading whatever happened to be in cache at the time is fine for our purposes.

3.2 Testing Method

When the benchmarks are run, two values are captured, the maximum synchronization cost and the minimum synchronization cost for each block. Figure 2.3 shows what the synchronization cost would look like from the scheduling of the various warps. The maximum time measures the actual duration of the synchronization barrier, which exposes many useful details about the memory subsystem performance at various loads. The minimum cost is the time after all the threads have synchronized before they all are

```
/*
 * Read some data from shared memory, then synchronize the warps
 */
__global__ void shared_mem_read_kernel( unsigned int *start, unsigned
    int *mid, unsigned int *end )
{
        __shared__ float data[ 1024 ];

        int idx = threadIdx.x + (blockDim.x * blockIdx.x);
        int idx2 = threadIdx.x;

        unsigned int start_reg, mid_reg, end_reg;

        // time we started mem op
        asm volatile("mov.u32 %0, %%clock;" : "=r"(start_reg));

        // read from shared memory
        float val = data[idx2];

        // time we started sync/finished mem op
        asm volatile("mov.u32 %0, %%clock;" : "=r"(mid_reg));

        __syncthreads();

        // time we finished sync
        asm volatile("mov.u32 %0, %%clock;" : "=r"(end_reg));

        start[idx] = start_reg;
        mid[idx] = mid_reg;
        end[idx] = end_reg;

        // use data to make sure read isn't optimized out
        data[idx2] = val+start_reg;
}
```

Listing 3.1: Shared read micro-benchmark

```
/*
 * Write some data to global memory, then synchronize the warps
 */
__global__ void global_mem_write_kernel( float* data, unsigned int
    *start, unsigned int *mid, unsigned int *end )
{
        // The index of the current thread
        int idx = threadIdx.x + (blockDim.x * blockIdx.x);

        unsigned int start_reg, mid_reg, end_reg;

        // time we started mem-op
        asm volatile("mov.u32 %0, %%clock;" : "=r"(start_reg));

        // memory write
        data[idx] = idx;

        // time we started sync/finished mem-op
        asm volatile("mov.u32 %0, %%clock;" : "=r"(mid_reg));
        __syncthreads();

        // time we finished sync
        asm volatile("mov.u32 %0, %%clock;" : "=r"(end_reg));

        // Save times to send back to the CPU
        start[idx] = start_reg;
        mid[idx] = mid_reg;
        end[idx] = end_reg;
}
```

Listing 3.2: Global write micro-benchmark

able to start running again. The minimum cost is mostly dependent on the time taken to schedule each of the warps again after the synchronization is complete. This cost is what is exposed in the actual runtime of the kernel, and what will need to be included in the model. The reason is because a block within a kernel finishes when the final warp in the block finishes, and the minimum synchronization cost is a direct cost that the slowest warp will experience, adding to its runtime. The synchronization also indirectly increases the memory cost since now many threads will all be requesting memory at the same time, rather than more staggered.

To measure these times the internal clock register in PTX is used. This register stores the count of elapsed cycles. By reading the register before calling `__syncthreads()` and after returning from the function call, the number of elapsed cycles can be computed and stored in global memory. Once the entire kernel has completed its execution, the calling C program retrieves the synchronization cost measurements from the GPU and filters out the needed values which are then saved to a file for later use.

Each kernel is run 10 consecutive times, and the minimum and maximum synchronization times are captured for each block of each kernel run. When the data are later analyzed all the values for the minimum and maximum are averaged for all 10 runs. The averaged values are then used to generate plots of the data to investigate how the costs vary over the number of blocks launched as well as by the number of threads which are in each block.

Although runs were done with block sizes ranging from 64 to 1024 threads per block, we will only show results for 256 threads per block for the sake of space. This size is a realistic size for the number of threads launched in a block in a real kernel. For smaller sizes, the effects are less pronounced and are therefore harder to see in the graphs.

3.3 Hardware

The machine specs on which the benchmarks were run is summarized in Table 3.1. The machine contained an NVIDIA C2075 Tesla card which uses the Fermi architecture. The Tesla series cards are designed specifically for use with CUDA rather than for use as a typical graphics card. The machine

GPU	NVIDIA C2075
Frequency	1.15 GHz
CUDA Cores	448
Memory Available	6 GB GDDR5
Memory Bandwidth	144 GB/sec

Table 3.1: Test machine GPU specifications

also has an older NVIDIA card which is used as the display adapter for the machine. This leaves the C2075 free from any other workload that may cause the results to vary significantly between runs.

Chapter 4

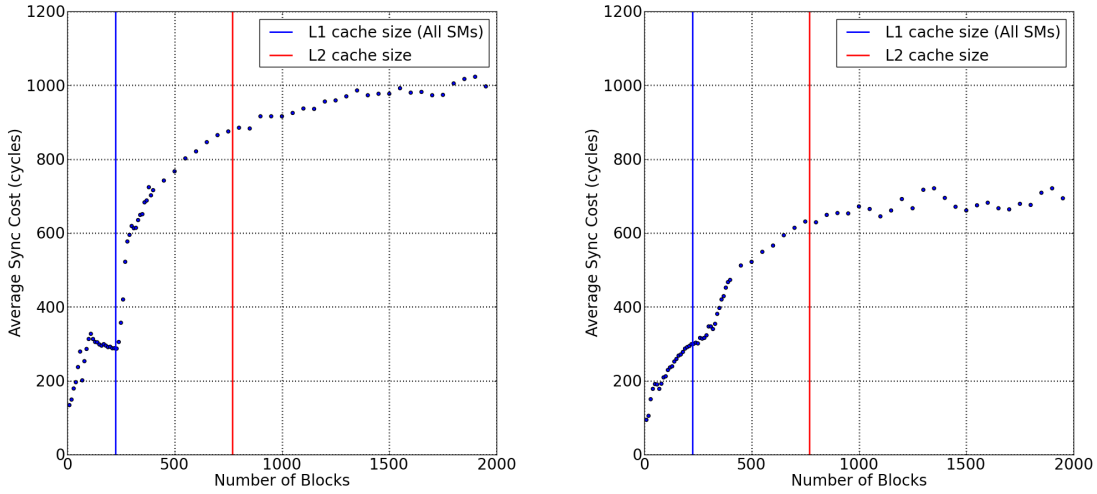
Results

This chapter presents the results obtained from the micro-benchmarks. The results are divided into two sections. The first contains the results from the global memory benchmarks, both for reading and writing, and for minimum and maximum synchronization cost. The second section contains the same results for the shared memory benchmarks. For the purposes of showing graphs which show as many features as possible the graphs for 256 threads per block were chosen. This value is large enough to very clearly show the various features while also being a value that may realistically be chosen for a real kernel in a program.

4.1 Global Benchmark Results

4.1.1 Maximum Synchronization Cost

Figure 4.1 shows how the maximum synchronization cost varies based on the number of blocks launched for a constant number of threads in each block when accessing global memory. The charts have lines showing when enough data are being used by the threads to fill up the L1 and L2 caches. One important thing to note about these figures is that the values are not the memory access costs, but the divergence in the execution of each of the blocks in the kernel caused by the memory accesses. In general the warps will be scheduled one after the other, each performing the same instruction. When all the warps ready to run another instruction have done so, it loops back to the first warp and starts again. However, some instructions, such as memory accesses, do not consume a constant number of cycles, and some



(a) Global read

(b) Global write

Figure 4.1: Maximum synchronization cost for 256 threads per block with global memory accesses.

threads will be skipped if they are still waiting on memory accesses. When a synchronization instruction is reached threads must wait for any threads that have executed more slowly than they did to catch up. The time between the first thread to reach the barrier and the last thread is what is measured by the graphs. The main driver of this in the micro-benchmarks is the differences in the time taken to access memory for the various threads.

Figure 4.1(a) shows the effect a global read has on the synchronization cost. For this micro-benchmark, as much data as can fit is first placed in the L1 and L2 caches. This more closely mimics a real kernel (not micro-benchmark) where the data would be present in some level of cache. The first vertical line on the chart shows the L1 cache size available for the entire GPU, and the second shows the L2 cache size. Almost immediately after filling the L1 cache, the synchronization cost begins to increase steeply after being at a slow decline for a few hundred blocks before. The L2 cache size has a much smaller effect on the synchronization cost, which reaches a more or less constant value after filling the cache.

There are three different regions on the chart. The first two fall in the L1 cache section. For small numbers of blocks launched, the GPU is not yet

fully loaded as more threads could still be run at the same time. As more threads are launched, the cost increases linearly with the number of threads. This trend continues until shortly after the maximum number of blocks that can fit in the GPU at any given time is reached; this is the peak followed by the slowly decreasing segment. For 256 threads per block with the 14 SMs available on the GPU used, this value is $\frac{1536}{256} \times 14 = 84$ blocks. In Figure 4.1(a) there is a split in the linear segment, most likely due to power saving features where not all SMs are active when the number of threads launched is low, as more threads are added more SMs become active. A similar, but less pronounced effect is seen in Figure 4.1(b) which is also likely due to more SMs being active with larger numbers of blocks being launched. By activating more SMs, the total available L1 cache is increased since each SM has its own set of L1 cache, so with more L1 there is more data that can be stored in L1 before it overflows into the L2.

After enough blocks are being launched to fill all the SMs on the GPU with extra blocks left over, those extra blocks begin to run after the previous blocks have finished. Since they are not started at the same time as a large number of other blocks, they encounter much less contention on the memory bus than the initial wave of blocks did, which is why the average cost is slowly decreasing in this area.

The final region on the plot is after the L1 cache is filled. At this point the graph takes on a logarithmic shape, increasing quickly to about 800 cycles before slowly increasing to 1000. The reason the cost grows so large is due to the large number of memory accesses to global memory that are occurring; if all of the accesses hit in L1 cache, then the total time to complete them all is much smaller than when some miss L1 and have to access L2, which is in turn much smaller than when the DRAM itself must be accessed. DRAM can be accessed in bursts; however there is still a finite throughput that it can accommodate which defines the lower bound on the time to read all of the required memory. The memory accesses for any given thread will be distributed throughout this range of time, and as it grows, so will the average time for an entire block to read all of the memory that it needs.

Figure 4.1(b) shows how a global write affects the synchronization cost. As with the global read, the L2 cache size appears to have little effect on

the synchronization cost, while L1 still appears to affect the average significantly. Unlike global read, this chart appears to have really only two regions.

The first region extends from the beginning until right around when L1 cache has been filled. This region is rather linear, leveling off slightly as L1 gets closer to filling up. Then, as was the case for global read, it begins to grow logarithmically, quickly increasing to 500 cycles before slowing and increasing to around 700.

Writing to global memory does not appear to be affected by having enough blocks to fill all the SMs in the same way as reading was. This is due to the caching policy, where writes are written to cache and flushed back to DRAM at a later time, which allows the blocks that launch in the second wave to enter into the queue faster than when reading, so they still encountered contention on the bus from the slower blocks in the first launch.

Other than looking at how the number of blocks launched affects the synchronization cost, we can also look at how the cost varies when different numbers of threads per block are launched. Figure 4.2 shows the synchronization cost as the number of threads in each block is changed. The synchronization cost for each thread per block value is that of the full 2000 blocks which ensures that the values shown are the steady state maximum synchronization cost.

Each of the vertical lines in the plots denote the switch over point from being able to run N blocks on each SM to being able to run only $N - 1$ blocks on each SM. For example, the line at 768 threads per block denotes where only one block can be run per SM to the right of the line and where two blocks can be run per SM on the left of the line. The lines would continue indefinitely; however below about 256 threads per block each data point lies in its own section. Taking the point with 128 threads per block, this results in $1536/128 = 12.0$ blocks running on each SM. Since 1536 happens to be evenly divisible by 128 there is no extra space left over, so when we increase the number of threads per block to 160 by adding another warp to each block, the new number of blocks that can fit in each SM is $1536/160 = 9.6$. This results in only 9 blocks running on each SM rather than the previous 12. The extra 0.6 means that the SMs will also have some

empty space that could be running more threads, but because it is not enough to fit an entire block it must be left empty.

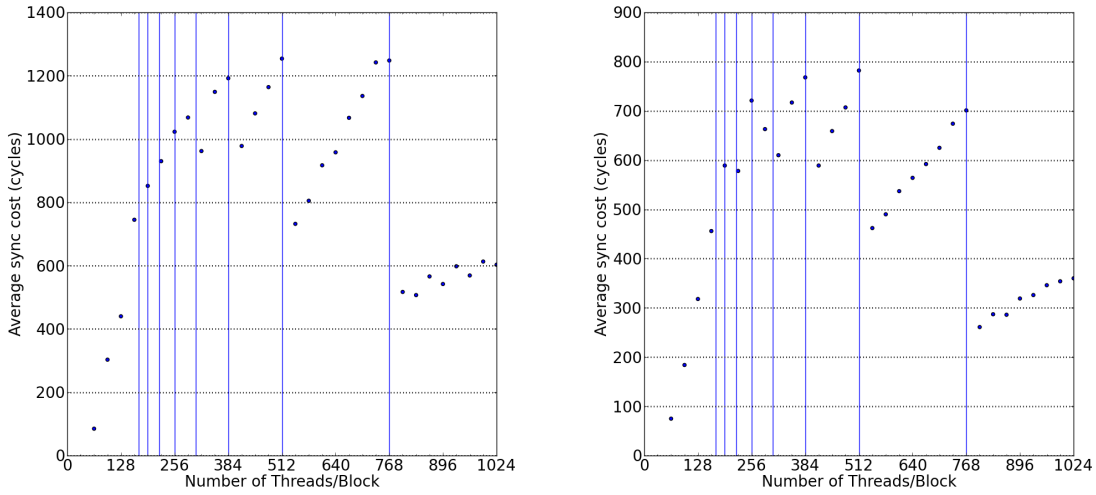
The data point immediately to the right of a line has the minimal occupancy in that section because there is just not enough extra space in each SM to fit one more block and the space must be left empty. As the data points move toward the next line, those empty spots in each SM begins to be filled in by the extra warps added to each existing block. Finally, when the next line is reached the entirety of the SM, or as close as it can be, is now being utilized, and the process repeats itself once another warp has been added to each block. This concept is called the occupancy of the SM and maximizing it is an important factor to optimizing the performance of a kernel.

The synchronization time drops when the occupancy drops because there are now fewer total threads accessing the memory system at any given time so the average time to read/write to the memory will be lower. Conversely, when the occupancy is higher, more threads are running and able to access the memory bus at the same time so there will be more contention which will result in longer wait times on average. The longer wait times lead to a greater discrepancy in when threads in each block actually get access to their memory. This discrepancy is the main driver of the maximum synchronization cost.

4.1.2 Minimum Synchronization Cost

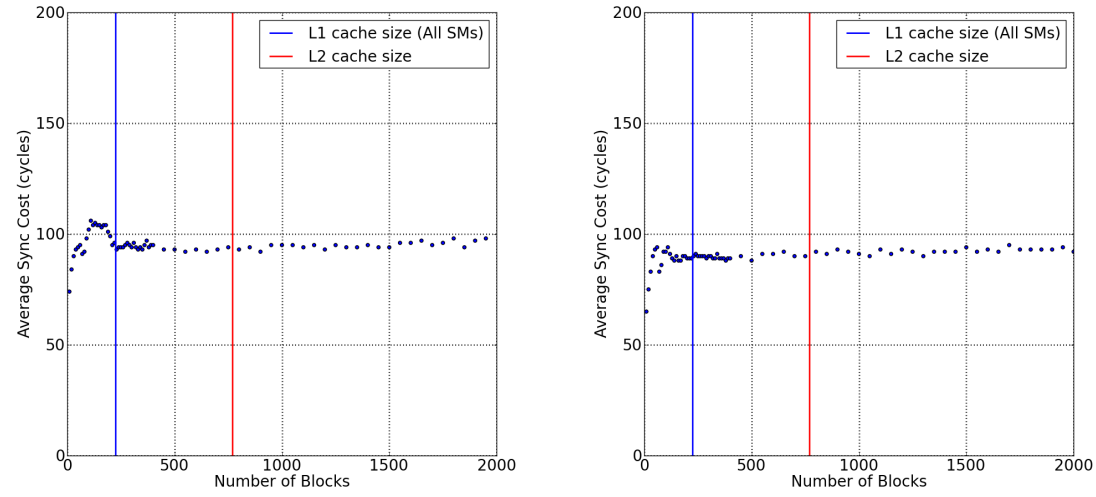
Figure 4.3 shows the results for minimum synchronization time of the global memory micro-benchmarks. These data are again for the value of 256 threads per block, so that it shows the same kernel launches as were investigated before.

In the figure, the lines for L1 and L2 caches are included as they were for the maximal cost; however, it can be seen that there is little effect on the data around these points. For the L2 cache line, there is no effect for either the read or write operation. The L1 cache line does seem to fall at the point where the read operation minimum synchronization cost levels off whereas the write is not affected. This makes sense because by default the global write operations follow a write-back cache scheme in the L2 cache, so the



(a) Global read (b) Global write

Figure 4.2: Maximum synchronization cost at 2000 blocks for various threads per block with global memory accesses.



(a) Global read (b) Global write

Figure 4.3: Minimum synchronization cost for 256 threads per block with global memory accesses.

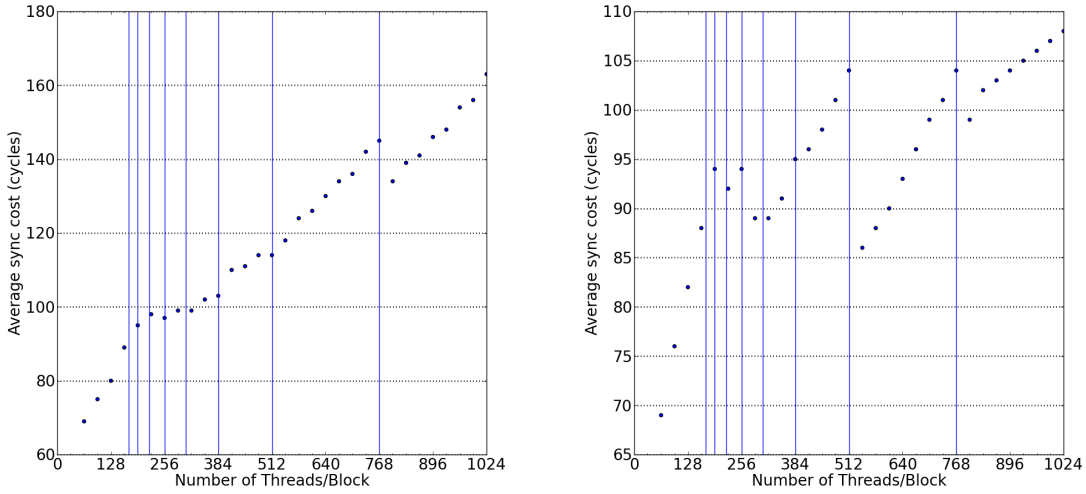
write operations should, for the most part, ignore the L1 cache. The same drop early in the data is seen again in Figure 4.4(b) once again this is likely the result of power saving features that cause more SMs to be activated only as larger numbers of blocks are launched.

The global read cost is slightly higher than it is for write. It also has a slight slope to the cost after it levels off whereas the global write cost levels and stays basically constant. The leveling period is due to the occupancy issues discussed previously; where a small number of blocks is run, there are not enough blocks to fill the entire GPU, so the costs that would arise from contention are not as high as they are when more blocks are run. After enough blocks are present to fill the GPU, the cost levels off. There is actually a small bump prior to where the cost levels off, which is where the occupancy is just reaching 100% of the total GPU. The initial launch of threads will all start executing at nearly the same time. The excess blocks that must wait until the first batch finishes get launched either by themselves or with a few other blocks. This reduces the total contention that is present in the system at the launch of a block on average. The bump which then decreases to the lower constant value is the result of this averaging taking place.

The global write seems to scale better in terms of the synchronization cost than the global read does. Its minimal cost is basically constant, having a very slightly upward slope, but the global read has a much higher slope to its cost as the value increases.

The minimal cost is much more affected by how many threads per block there are than it is by how many blocks are run because it depends on the number of warps that can fit on each SM at any given time. The value plotted for each point is the value at 2000 blocks, which ensures that all the data are from where the graphs were at the steady state costs.

Figure 4.4 shows the effect on minimum synchronization cost for global memory accesses as the number of threads per block is changed. The data are similar to the maximum cost; however the rate of increase within each section is much lower, and overall the value tends to be much more constant. The first few threads per block sizes increase sharply because at this point there are so few threads per block that a block consists of just a few warps,



(a) Global read

(b) Global write

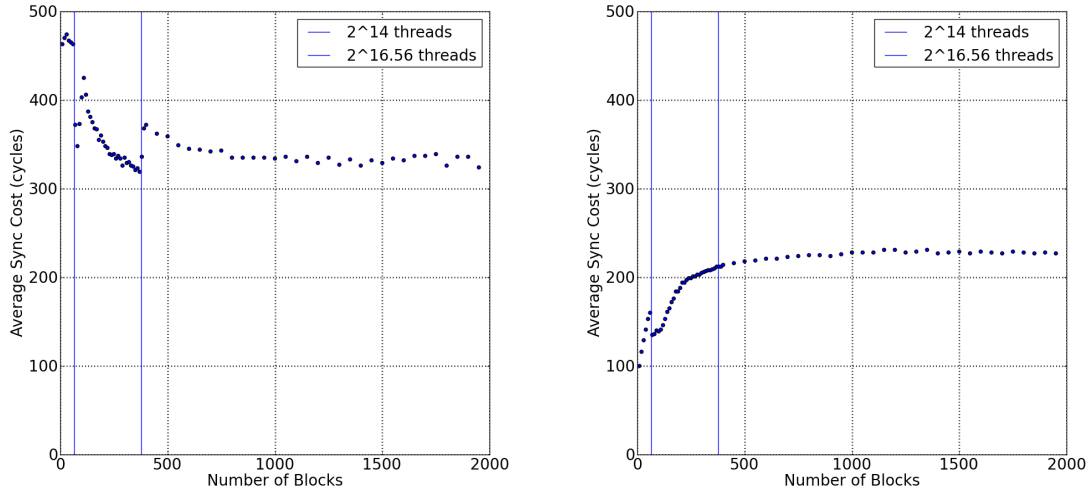
Figure 4.4: Minimum synchronization cost at 2000 blocks for various threads per block with global memory accesses.

and the average time for the few warps to be scheduled is low. As the number of threads per block is increased so is the number of warps per block, and it begins to take longer on average to schedule all the warps. After around 200 threads per block the value levels off at around 100-120 cycles where it remains until it encounters the two and one block per SM range. The value for global write is slightly lower than the value for global read which would indicate that the global write kernel is slightly more efficient at scheduling the warps after the synchronization occurs.

4.2 Shared Benchmark Results

4.2.1 Maximum Synchronization Cost

The global memory operations are both rather similar to one another, in terms of the general shapes of the graphs and the stages that the cost goes through to make that graph. Shared memory is different because it shares physical cache as the L1 cache. Shared memory is explicitly managed and is not backed by any higher levels of cache like the L1 cache is. By default,



(a) Shared read

(b) Shared write

Figure 4.5: Maximum synchronization cost for 256 threads per block with shared memory accesses.

the first level of cache is split: 16 KB for L1 cache of global memory, and 48 KB for shared memory.

Figure 4.5(a) shows how shared read affects the synchronization cost. It is the most unique of the micro-benchmark data because rather than increasing asymptotically toward some value, it actually starts high, then decreases asymptotically toward a steady state. All of the micro-benchmark data have a sudden drop in the synchronization cost at exactly 16384 threads launched. In the other charts, the drop is less noticeable, or the cost quickly increases back to its original trend. This drop is likely due to power saving features of the architecture. When fewer threads are launched, some SMs are inactive, and as the numbers of threads rise, more SMs are activated. With shared read, the drop is larger and more persistent than in the other micro-benchmarks, and there is an accompanying rise in the data later on where it suddenly spikes upward. This second point again seems to depend on the number of threads launched, though there is some variation with different numbers of threads per block. This rise could be due to reaching the limit of occupancy similar to the global memory micro-benchmarks. The memory access times were also extracted in the micro-benchmarks using the clock

register. At the point where the decrease occurred, a small, short-lived increase in the average read time was seen in all of the micro-benchmarks. The micro-benchmarks which more strongly show the drop had a sharper increase in the memory time. Ignoring the area where the data is dropped lower, the shared read seems to have only a single asymptotically decreasing region, unlike the global memory operations where multiple regions were present. In the area where the drop occurs, the data seem to follow the same curve, just displaced lower.

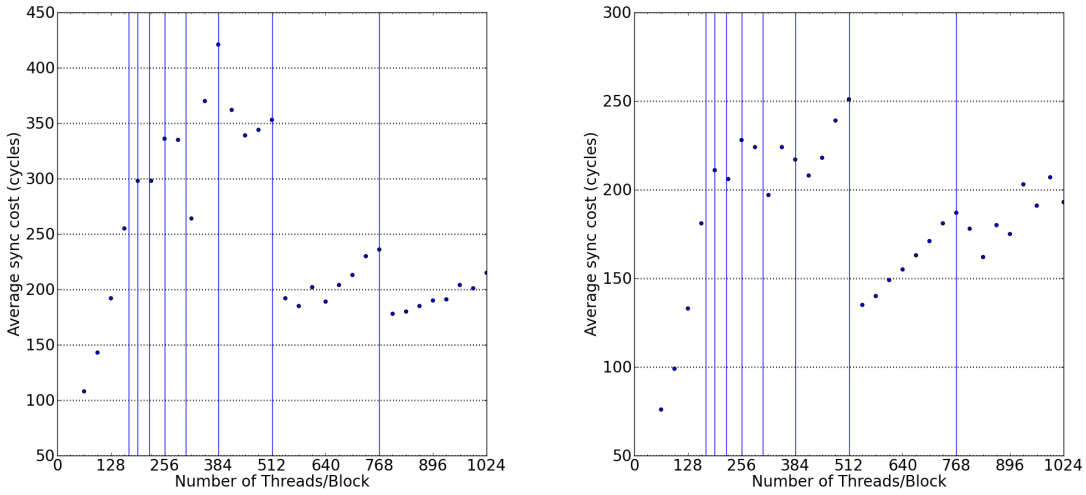
Figure 4.5(b) returns back to the pattern seen for the global memory operations, where the synchronization starts off low, and asymptotically increases toward its steady-state value. As with the shared read, there is only the single region present due to the lack of any other levels of cache in the shared memory system.

Figure 4.6 shows how the cost is affected by changing the number of threads in each block. The effect is the same as it was with global memory where the cost is lowest when the occupancy is lowest and rises in value until the next transition region. The unique shape of the shared read graph seen before is not present in these graphs, and both sets of data are very similar showing that the strange features are present only for lower numbers of blocks launched.

4.2.2 Minimum Synchronization Cost

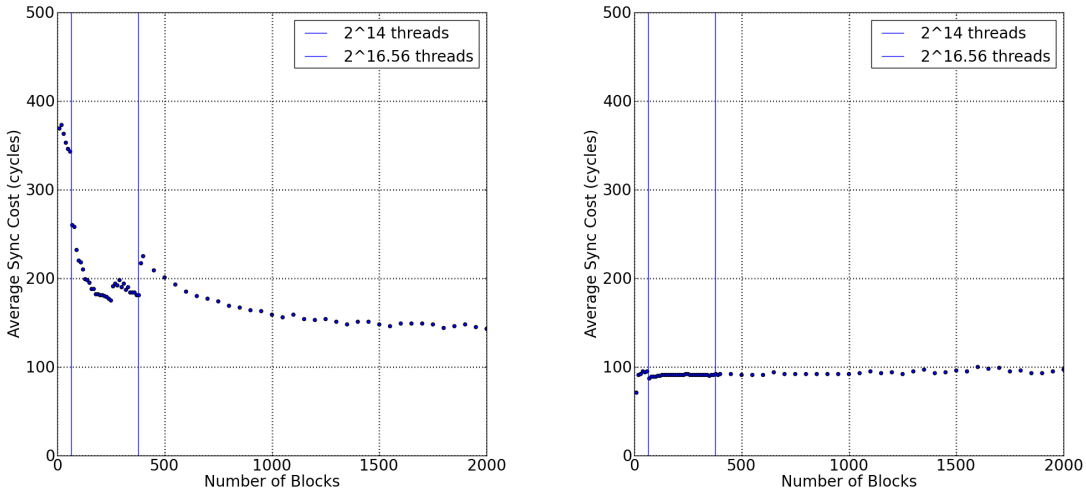
Figure 4.7 shows the minimum synchronization cost for shared memory accesses. Similarly to global, the markers for the location of the jumps in the data were left in. Shared write has the same drop as with the maximum cost; the shared read however, has the same drops as the maximum cost, but also introduces another drop between the two original drops.

The shared memory costs are much closer to in terms of the difference between their maximum and minimum values than global memory was. Shared read starts out at around 450 cycles and decreases to a little above 300 cycles for the maximum cost. The minimum cost starts out at around 350 cycles and decreases down to about 150 cycles. This puts the difference between maximum and minimum cost at only around 150 cycles rather than



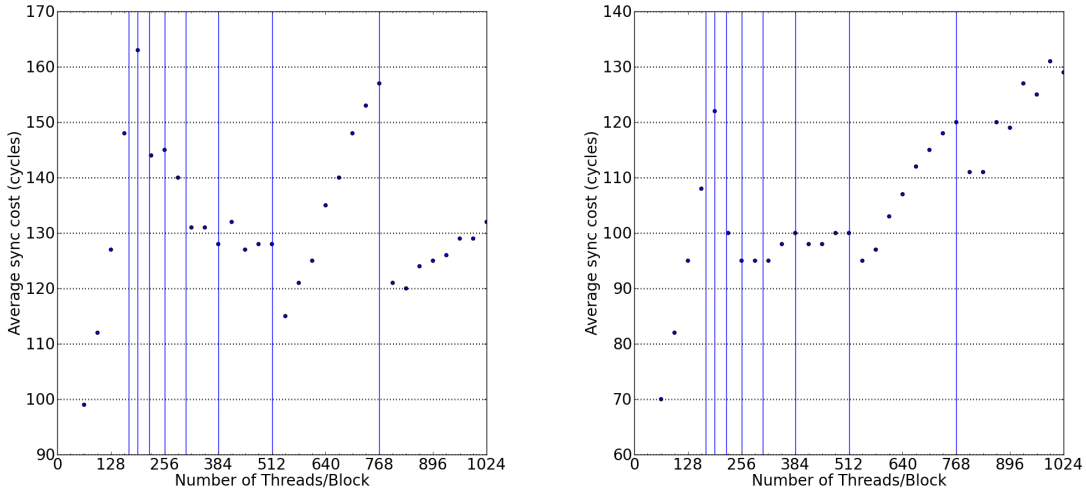
(a) Shared read (b) Shared write

Figure 4.6: Maximum synchronization cost at 2000 blocks for various threads per block with shared memory accesses.



(a) Shared read. (b) Shared write.

Figure 4.7: Minimum synchronization cost for 256 threads per block with shared memory accesses.



(a) Shared read.

(b) Shared write.

Figure 4.8: Minimum synchronization cost at 2000 blocks for various threads per block with shared memory accesses.

1000 cycles. This shows how much better shared memory is at minimizing the differences in memory access times since it resides entirely within the L1 cache.

Though shared memory provides memory accesses that have much less variance in the value, it is more complicated to use. Shared memory is an explicitly managed L1 cache, so any memory values to be stored in the shared memory must be placed there manually; also, any values that need to be placed back into global memory as the result of a computation need to be placed there manually.

The shared write values are actually very close to the global write values in terms of shape and the value of the data. Both level off relatively quickly once the GPU is completely full of blocks to run, and both remain essentially constant at a cost of around 100 cycles.

The shared read is the odd one out in the set of micro-benchmarks that were run; it starts at a high cost initially and drops as more blocks are run toward a value of around 150. The decay that occurs in the graph is likely due to the averaging effect of adding more and more blocks that happen to experience lower costs, so the outliers that were measured initially begin to

diminish in their effect on the data. There are a couple of points in Figure 4.7(a) where the cost jumps back up again, which seem to be related to how many threads in total were launched. This probably means that the outliers that cause the cost to start high rather than low, happen to reappear at these points only to be averaged away again as more blocks are added.

The effect of changing the number of threads per block on shared memory is very similar to that of the global memory. The initial cost is low and rises as more threads are added to each block, plateauing at about 200 threads per block. After this the value remains rather steady at between 100 and 140 cycles for write and read respectively.

Chapter 5

Improved GPU Model

In this chapter, the synchronization cost determined by the micro-benchmarks is explained and tested using a GPU model. The model used is from [8] with some modifications. The original model, with its model for the synchronization cost, is compared with the improved equations presented in this chapter. In all cases, the original model overestimates the synchronization cost, and while the improved equations are not perfect, they allow the model to more accurately predict the runtime.

5.1 Model Changes

There were a few changes that needed to be made to the model to improve the results. One thing that the model did not correctly account for was the ability to schedule multiple instructions at the same time in each SM. $W_{parallel}$ is the time that it takes each SM to schedule all of the instructions for all of the blocks that it is responsible for running. The original version assumes that only one instruction is able to be scheduled per cycle. On the Fermi cards there are two schedulers that are capable of effectively issuing 2 instructions per cycle. The newer Kepler cards have larger SMXs which contain four schedulers that are each capable of issuing two instructions at a time, so in the interest of making the model as forward compatible as possible the equation will be modified to contain a card specific parameter for total instructions able to be issued per cycle. The original equation is reproduced in (5.1), and the modified version is shown in (5.2).

$$W_{parallel} = \frac{N_{insts} \times N_{total\_warps}}{N_{active\_SMs}} \times \frac{avg\_inst\_lat}{ITILP} \tag{5.1}$$

$$W_{parallel} = \frac{N_{insts} \times N_{total\_warps}}{N_{active\_SMs} * issue\_size} \times \frac{avg\_inst\_lat}{ITILP} \tag{5.2}$$

The new $issue\_size$ parameter is equal to two for Fermi and eight for Kepler and effectively scales the number of SMs available. This accounts for the fact that each is capable of running more than one instruction per cycle. Without the change to the $W_{parallel}$ equation, the times are higher than the actual runtime even after the change for the synchronization is taken into account. The value for $W_{parallel}$ is a component of the computation time, T_{comp}, along with the serial time W_{serial}.

The change to the synchronization equation results in a much simpler equation since within the general working area for kernels the minimal synchronization cost, which is the one that actually extends the runtime of the kernel, is essentially a constant value. Generally immediately after a synchronization a read from memory will occur, which means that the bottleneck is going to be the load/store units in the GPU. The Fermi architecture has only 16 load store units for a warp size of 32 threads, which means that each thread will take 2 cycles to schedule its read. At maximum capacity each SM can handle 48 warps. When it is at full capacity multiplying the two values would give us a cost of 96 cycles, which is very close to the 100 cycles seen in the benchmarks. However, when fewer than 48 warps are running the cost will be lower. This fits with the lower values of the minimum synchronization cost seen in the results of the micro-benchmarks when low numbers of blocks were launched. The original equation for the synchronization cost is shown in (5.3), and the improved version is shown in (5.4).

$$F_{sync} = \Gamma \times avg\_DRAM\_lat \times \frac{N_{mem\_insts}}{N_{insts}} \tag{5.3}$$

$$F_{sync} = N_{warps\_per\_SM} * \frac{warp\_size}{N_{load\_store}} \tag{5.4}$$

As can be seen, the original model for the synchronization cost had a direct relationship with the DRAM latency, which would seem to imply that the equation may be modeling the maximum synchronization time rather than the minimum. The maximum cost seemed to be affected much more strongly by the memory system. The minimum cost is what should be modeled because a block within the kernel is finished once the last warp finishes, and the minimum cost is a direct part of its execution time as shown in 2.3. The maximum cost does effect the overall time, but only indirectly through increased memory access time since all the threads will now be requesting memory at nearly the same time, rather than being more spread out. The O_{sync} equation remains unchanged since it merely scales the F_{sync} cost to account for all of the synchronizations that occur in the kernel.

All of the other model equations do not change from their form in the original model, see Section 2.2.3.

5.2 Micro-benchmark Modeling

To get a quick idea of how much more accurate the improved model will be with the updated equations, the model was applied to the micro-benchmark kernels used previously. One thing to note about these kernels is that they do not represent a typical CUDA kernel workload. They perform very little computational work and have a very short runtime. They do however work well for performing measurements of different aspects, which is what they were designed for. Although the error with the improved model is still somewhat larger than desired, it does not mean that this error will carry over to other, more typical, kernels.

5.2.1 Global

The data in Figure 5.1 show the error for the global read kernel with the original model and the improved version. In the figure, a value of 100% means that the modeled runtime is double the actual runtime. As can be seen, the original model has an error on average of around 3000% which

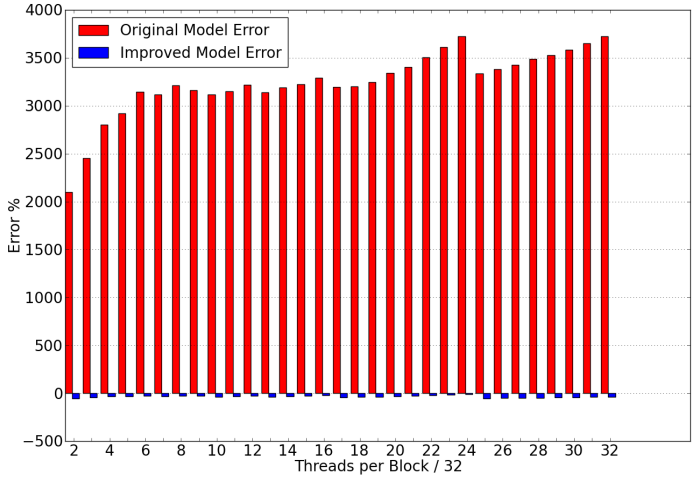


Figure 5.1: Average % error comparison of the original model and the improved model for the global read kernel.

means that the modeled runtime is about 30x larger than the actual runtime. The improved model performs significantly better than the original, and now actually underestimates the runtime by around 30%, an improvement of 100x.

Note how the error is mostly constant through all the different threads per block sizes. This is because for the smaller numbers of threads per block, where the minimum synchronization cost was lower, is modeled by including the N_{active_warps} term in the equation for synchronization cost. The smaller numbers of threads per block will have fewer warps for any given number of blocks launched.

Figure 5.2 shows the % error for global write for the original and improved models. The original model did a slightly better job with this kernel than it did with the global write kernel; this time the average error is about 1200% less than half the error in the global read kernel. However this still means that it predicts a runtime of about 13x larger than the actual runtime.

The improved model once again performs significantly better than the original model. The average error with this kernel was around 40% which is slightly worse than the global read; however it is still a 30x improvement over the original model.

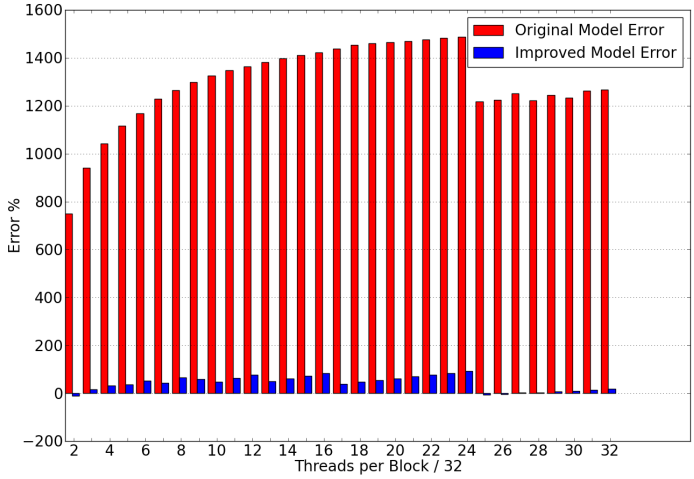


Figure 5.2: Average % error comparison of the original model and the improved model for the global write kernel.

5.2.2 Shared

The improved model performs similarly for the shared benchmarks as it did for the global benchmarks. The original model is the most inaccurate for these kernels with the average error falling between 5000% and 6000% which corresponds to about 50x to 70x larger than the actual runtime of the code.

Figure 5.3 shows the results for the shared read kernel. The original model overestimates the cost by around 50x for this kernel. With the improved model, the accuracy improves to around 30% or 0.3x longer than the actual runtime. The improved model actually both over and under estimates the runtime at various points. The previous global kernels either over or underestimated consistently.

Finally, Figure 5.4 shows the results for the shared write kernel. The original model performed worst for this kernel with an error of around 6000%, or 60x overestimation. The improved model performs almost identically to the shared read results, with an error of about 30% on average. It also both over and underestimates the runtime similarly to how the shared read did.

As can be easily seen, while the errors in the micro-benchmark kernels may not be as low as desired, they are still significant improvements over

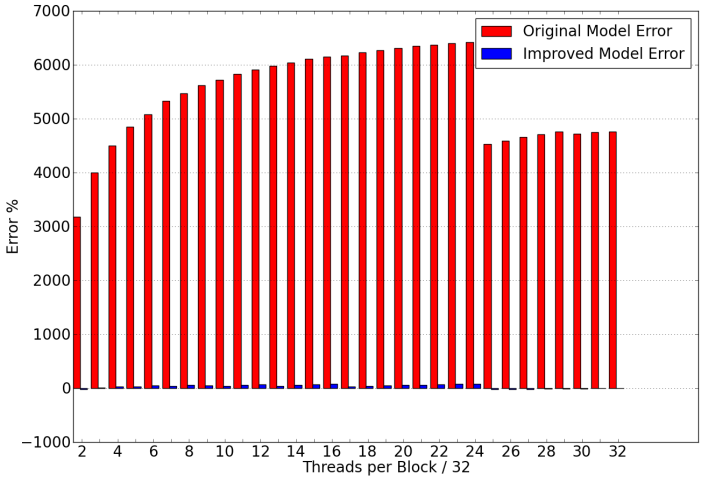


Figure 5.3: Average % error comparison of the original model and the improved model for the shared read kernel.

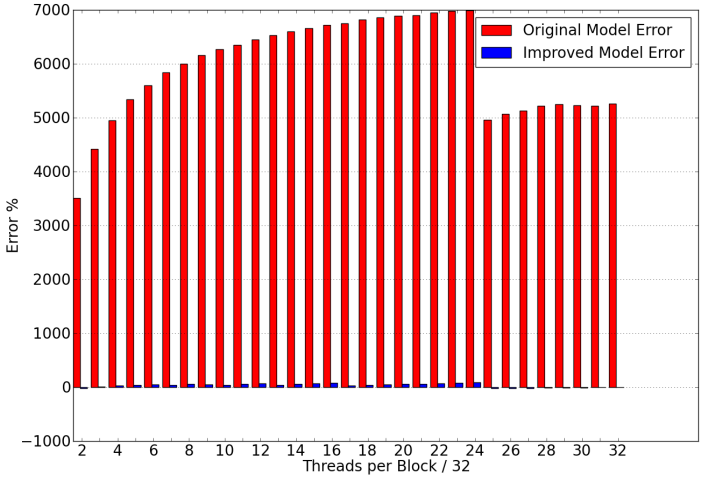


Figure 5.4: Average % error comparison of the original model and the improved model for the shared write kernel.

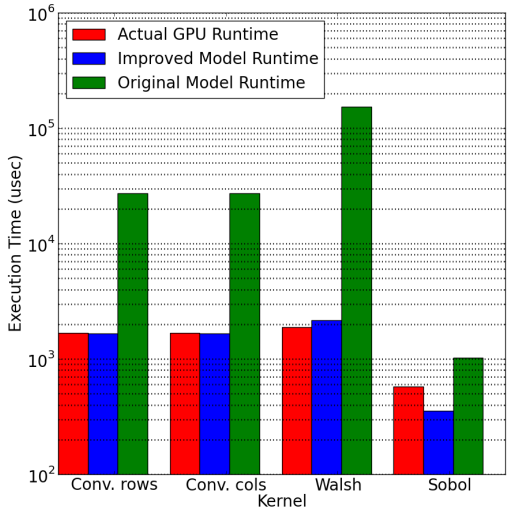


Figure 5.5: Actual vs Modeled, with and without synchronization cost, for various kernel workload types.

the original model. A better test of the performance of the model is to apply it to a kernel with a more typical workload.

5.3 Modeled Kernels

Figure 5.5 shows the actual runtime on a Fermi GPU, the modeled runtime with F_{sync} from the original model, and with the improved F_{sync} model. The costs are shown in microseconds on a logarithmic scale.

The kernels used to test are all part of the CUDA GPU SDK example kernels. *Conv. rows* and *Conv. cols* are the kernels that are part of the *ConvolutionSeparable* example in the SDK. These kernels are very similar, hence the nearly identical runtime each has. With the original model the runtime is significantly higher than with the improved equation and the actual runtime on the GPU. For the original model case, the cost is dominated by the computational cost, which includes synchronization; however, with the improved model, the cost is dominated by the memory costs as the synchronization cost is much lower than the original model. The computational cost of the model is the time taken to schedule all of the instructions

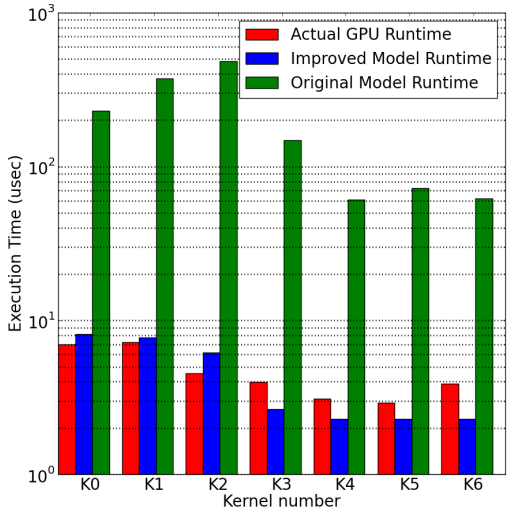


Figure 5.6: Actual runtime vs modeled cost, with and without synchronization cost, for reduction kernels.

in a kernel plus all the overhead associated with those instructions, with the exception of memory access times.

The second kernel is the batch1kernel from the *FastWalshTransform* example. This kernel is dominated by the computational cost in both the original and the improved model. This results in a much larger error in the original model because the difference in actual cost is not bounded by the memory access cost as it was with the convolution kernels, so all of the error is exposed.

Next, the *Sobol* kernel was modeled. This kernel isn't dominated by either the computation nor the memory cost, each cost is partially hidden. In this case the error of the original model is smallest.

Finally, the *Reduction* kernels were modeled, as shown in Figure 5.6. These kernels were chosen because they are an iterative example of the optimizing a kernel, starting from the naive first implementation, and moving to a much more optimized final kernel. These kernels have a much smaller runtime than the kernels in Figure 5.5, so the errors exposed by the synchronization cost is a much larger percentage of the overall runtime than the longer kernels discussed previously. The same pattern holds with the reduction kernels as with the previous ones. The original model overestimates the

runtime of the kernels, and the improved model tends to underestimate the runtime.

The improved model is better at predicting the actual runtime of a kernel than the original synchronization equation was. The error tends to stay below about 15%; however there are some kernels, such as the Sobol kernel, where the model itself does not perform as well, so the error is higher.

5.3.1 Kepler vs Fermi

The previous work focused on the Fermi architecture of NVIDIA GPUs as this was the architecture targeted by the original model used. Since [8] was published, NVIDIA released a new architecture named Kepler. Kepler adds significantly more computational hardware to each SM along with many other changes to the overall architecture. With Fermi, the improved equation for the synchronization cost gives a value of 96 cycles when at full occupancy. For Kepler, the number of load store units was doubled, to 32, and the maximum number of warps each SM can run was increased, to 64. Doubling the load store units should result in roughly halving the synchronization cost. The equation predicts that the minimum synchronization cost for Kepler should be $64 \times \frac{32}{32} = 64$ cycles.

Figure 5.7 shows the results of the shared read micro-benchmark with 256 threads per block. The actual results are lower than the predicted values; however with all the other changes to the architecture in Kepler, it is possible that some of these other changes are involved in this discrepancy. It can be seen that Kepler does appear to follow what would be expected with a significantly lower synchronization cost due to the addition of more load store units in each SM.

Another important thing to note about these Kepler results when compared to the Fermi results is that the shared memory read micro-benchmark had results that did not match very well with the other three micro-benchmarks. For the Kepler results, the behaviour seen in the shared read in Fermi has disappeared, and it now follows the same pattern that the other benchmarks did. It would seem that the drops and rises in the synchronization cost was, at least partially, related to the way Fermi handled the shared read memory

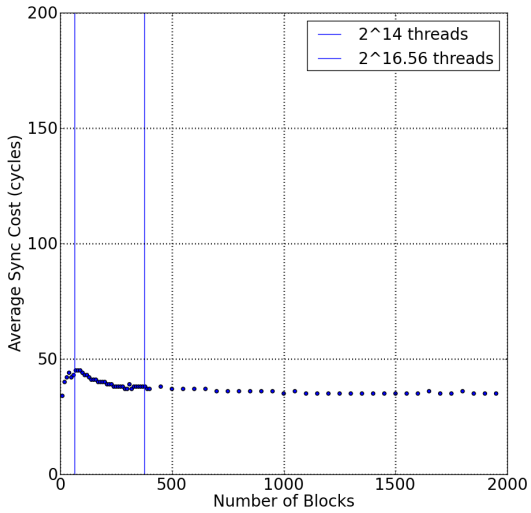


Figure 5.7: Minimum synchronization cost for shared read on the Kepler architecture

accesses. The Kepler architecture appears to fix the issue.

Chapter 6

Conclusion

The original version of the model works relatively well at predicting the runtime of CUDA kernels running on the Fermi architecture; however, it falls short when the kernel involves any calls to the `__syncthreads()` function. Not all kernels will require the use of synchronization between threads, but if any data need to be shared within a block the threads must be synchronized.

Through the use of micro-benchmarks, the effect of different types of memory accesses on the synchronization cost was investigated. Both the *maximum synchronization cost* and the *minimum synchronization cost* were analyzed, each one gave insight into different aspects of the GPU architecture. The *maximum synchronization cost* gives insight into the memory system showing how different kernel launch sizes effect the access time for memory instructions. This cost does not directly add to the overall kernel runtime, but instead it increases the memory access cost by causing memory accesses that would have occurred in a more staggered manner over time to occur much closer together. The *minimum synchronization cost* is the cost that directly adds to the execution time of the kernel. It is mostly influenced by the scheduling of warps in each SM, since once all threads have been synchronized they cannot start running another instruction until the SM gets around to scheduling it once again.

The original model's equation for the synchronization cost appears to model the *maximum synchronization cost* rather than the *minimum synchronization cost* which should have been modeled. By developing a model for the *minimum synchronization cost* to be used in the model, the accuracy of the model was greatly improved for kernels that contain a synchronization

cost. The improved model has an error for more general kernel workloads of around 15%, though sometimes higher due to other errors in the model from simplifications of the architecture.

By having a more accurate model of the kernel runtime allows it to be used for more applications, such as simulating scheduling of a heterogeneous system or to determine which GPU is required for a desired level of performance. This is useful as it can save money by allowing the cheapest GPU that will work, either for a single kernel, or for an algorithm that will run on a heterogeneous system.

The synchronization cost was also tested on the Kepler architecture to show that it applies to more than just the Fermi architecture. One of the main components of the *minimum synchronization cost* is the number of load store units that each SM has. Kepler has double the number that Fermi does, the equation would predict that by doubling the number of load store units the *minimum synchronization cost* should halve, and the results obtained roughly correspond to the predicted values.

Bibliography

[1] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator". In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174. Ieee, April 2009.

[2] Aniruddha Dasgupta. *"CUDA performance analyzer"*. M.s. thesis, Georgia Institute of Technology, 2011.

[3] G.F. Diamos and A.R. Kerr. "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems". In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364, 2010.

[4] Sunpyo Hong and Hyesoon Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness". *ACM SIGARCH Computer Architecture News*, 37(3):152, June 2009.

[5] V. Jimenez, L. Vilanova, I. Gelado, and M. Gil. "Predictive runtime code scheduling for heterogeneous architectures". *High Performance Embedded Architectures and Compilers*, pages 19–33, 2009.

[6] Kishore Kothapalli and Rishabh Mukherjee. "A performance prediction model for the CUDA GPGPU platform". *International Conference on High Performance Computing (HiPC)*, pages 463–472, December 2009.

[7] Jacques A. Pienaar, Anand Raghunathan, and Srimat Chakradhar. "MDR: performance model driven runtime for heterogeneous parallel

platforms". In *Proceedings of the International Conference on Supercomputing*, pages 225–234, 2011.

[8] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. "A performance analysis framework for identifying potential benefits in GPGPU applications". *ACM SIGPLAN Notices - PPOPP '12*, 47(8):11, September 2012.

[9] Yao Zhang and J.D. Owens. "A quantitative performance analysis model for GPU architectures". *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.